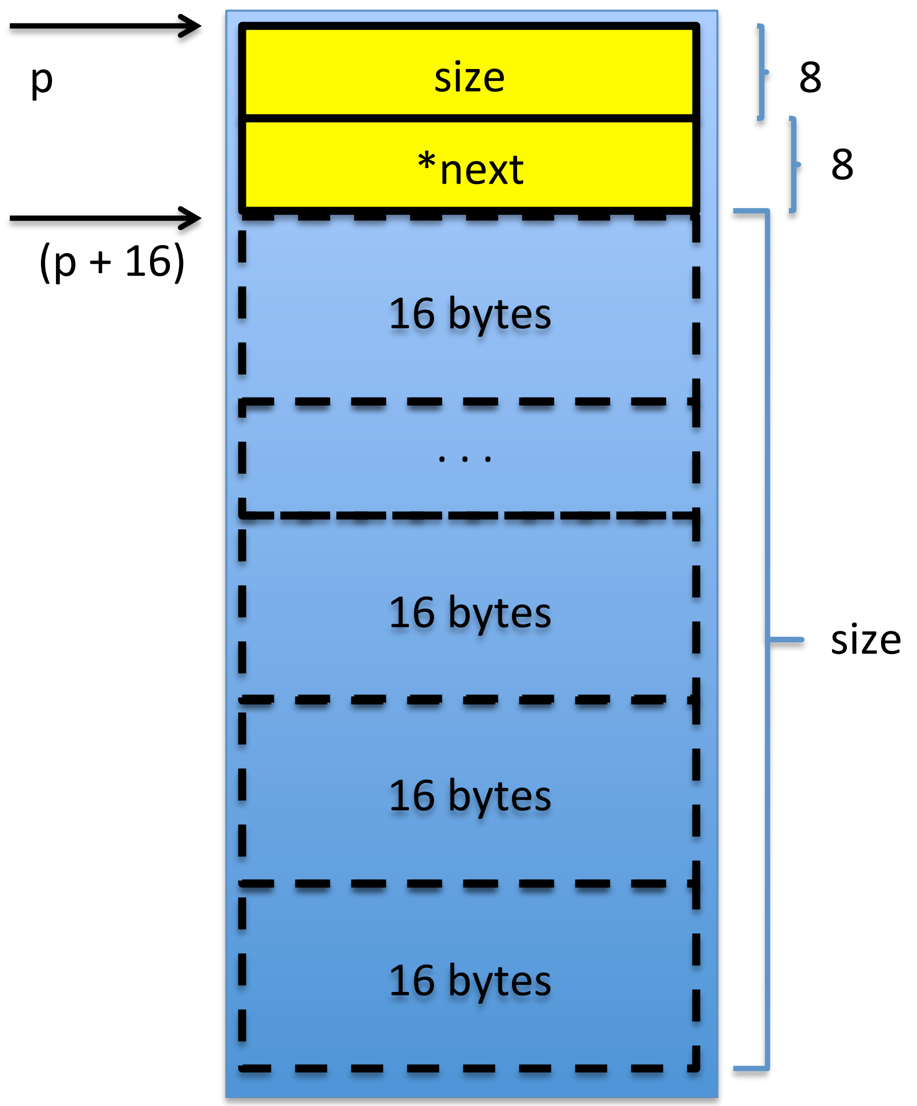
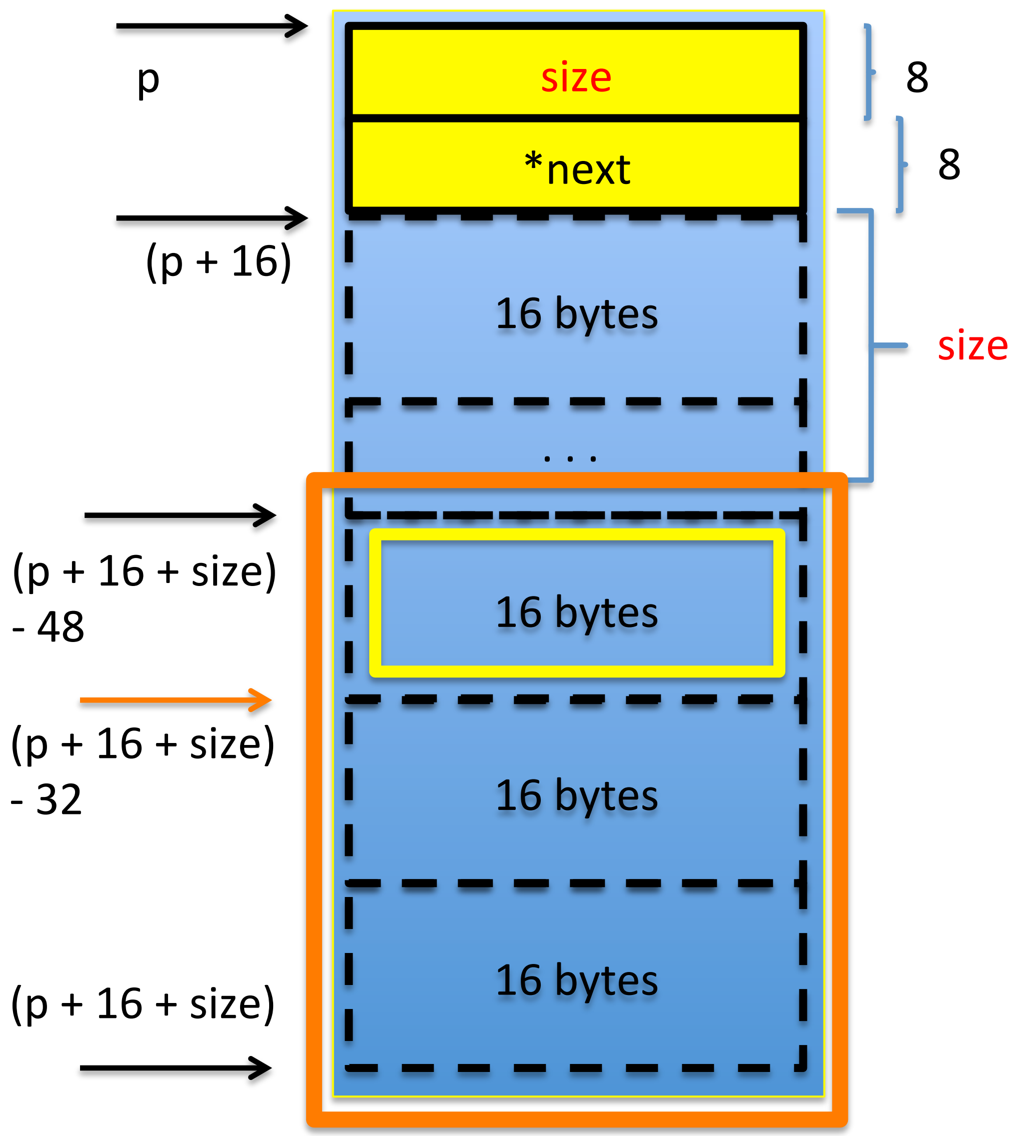
getmem if free_list is empty

- First, call malloc to add a large block of memory to the free_list (blue shading), including room for the 16-byte header block containing size and *next (yellow shading)
- malloc returns p
- getmem returns (p + 16) assuming we use this entire block
- The value of size is a multiple of 16

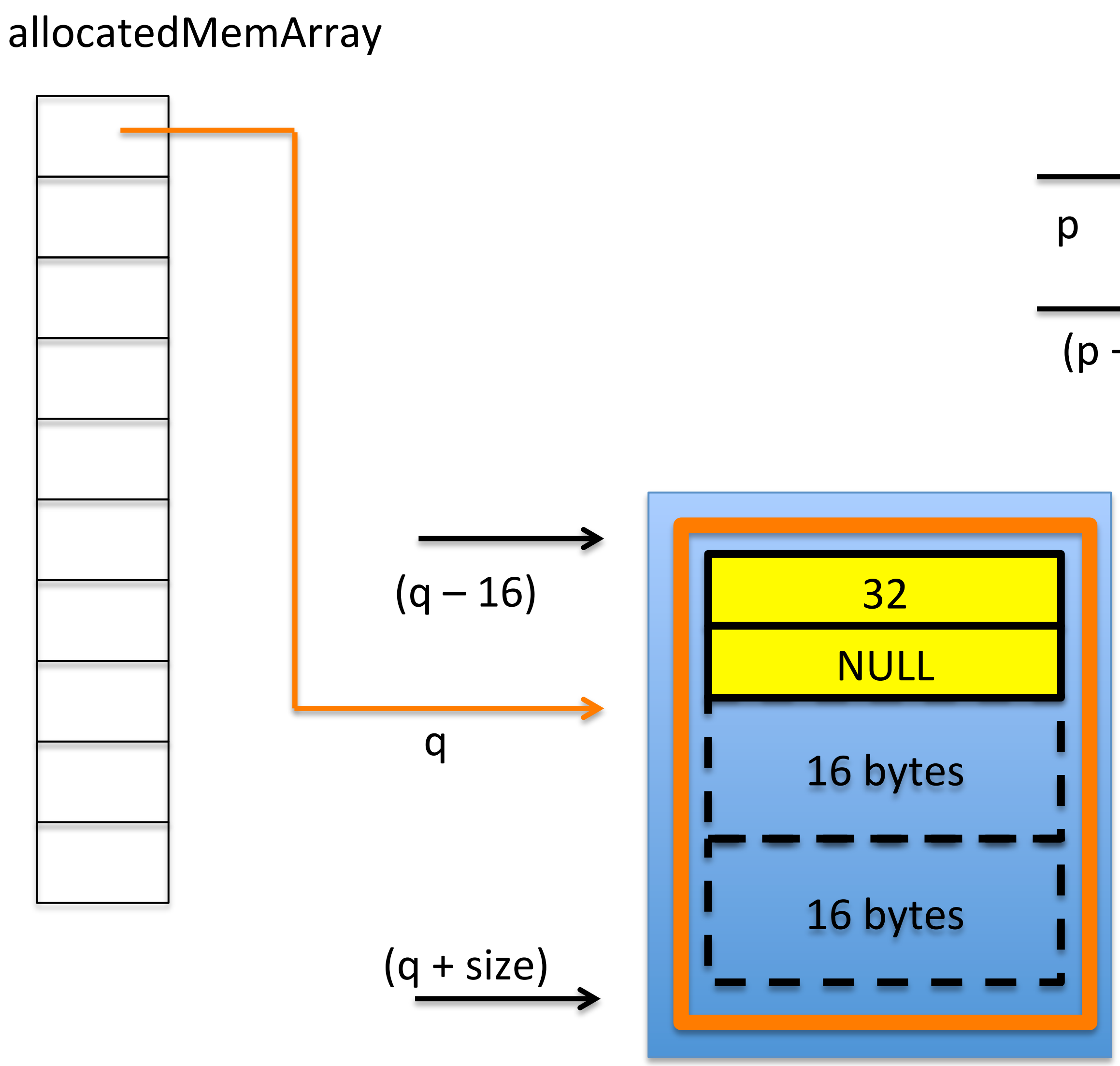


If not using a whole block

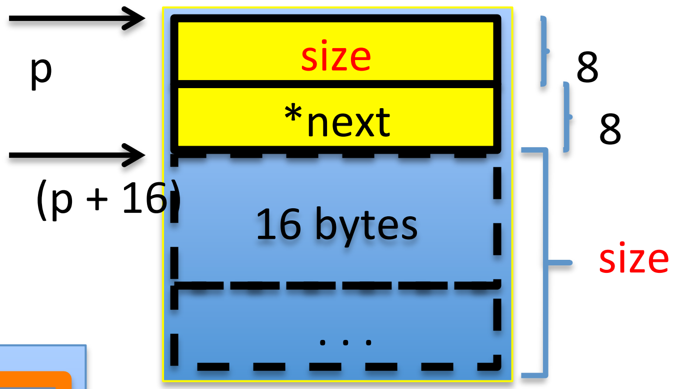
- If a call to getmem finds a large enough block already on the free_list and doesn't want to use it all, we can split the block
- Example, maybe we want 20 bytes
- Need to return an address that is a multiple of 16 so will need to pad 20 to 32, and don't forget another 16 for a header, for a total of 48. Return orange pointer from getmem.
- Adjust size on the block portion still on free_list to (size – 48)



Where do blocks "allocated" by getmem go?

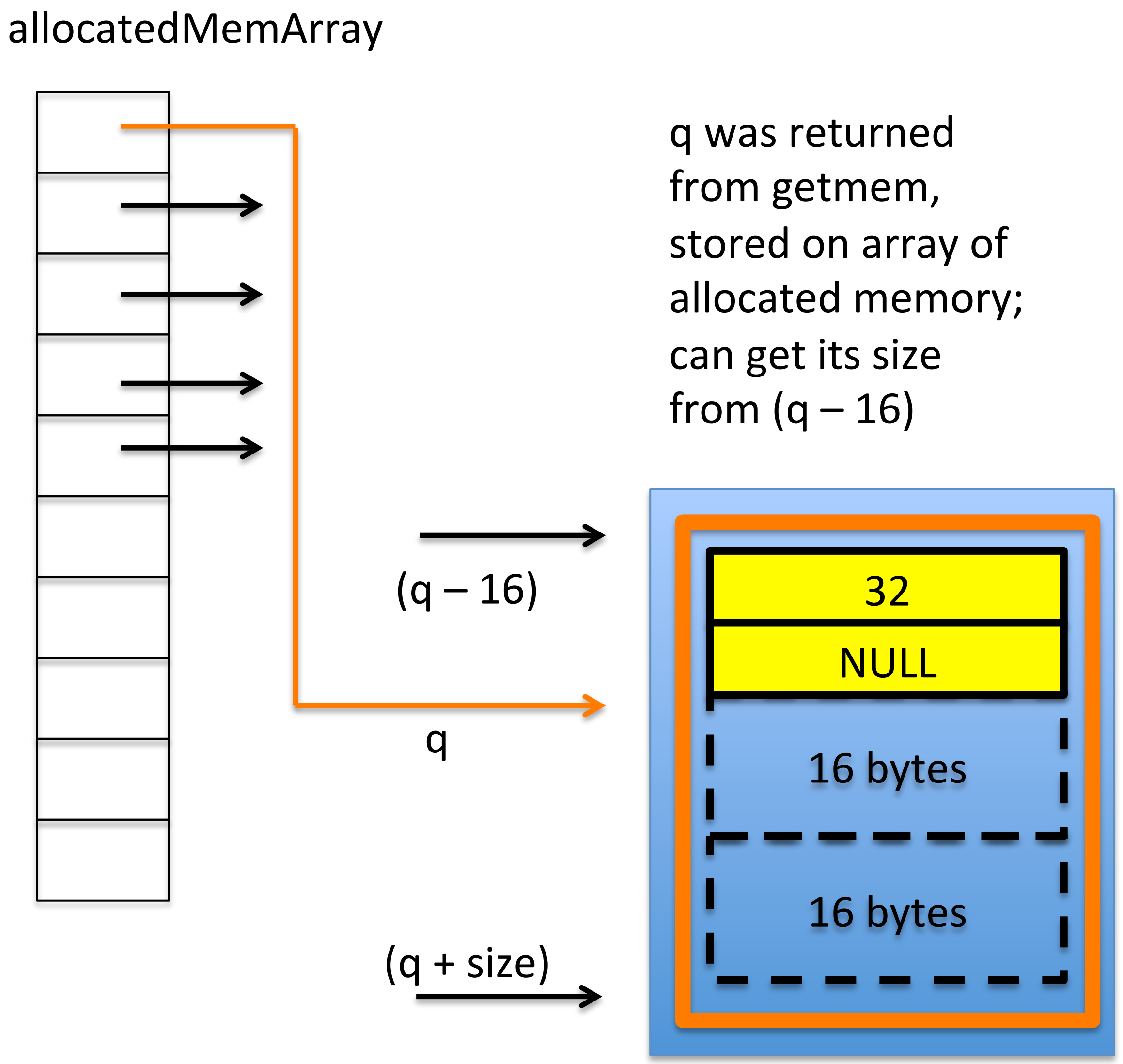


Example from previous slide, part still on free_list



q was returned from getmem, stored on array of allocated memory; can get its size from (q – 16)

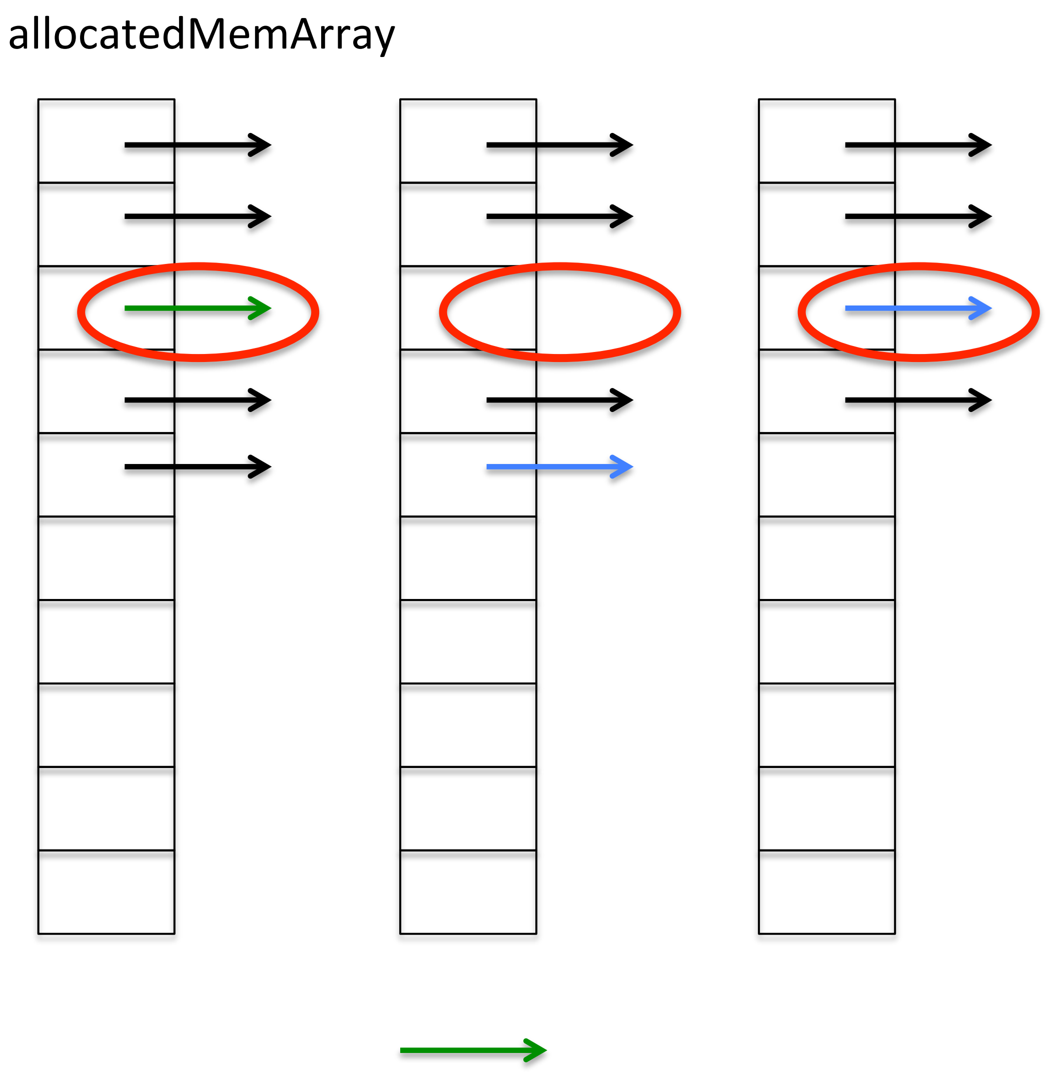
allocatedMemArray



q was returned from getmem, stored on array of allocated memory; can get its size from (q – 16)

Holds pointers returned by getmem function. Maximum size of n_trials. Fill from position 0 onwards (entire array may not ever be full). The freemem function randomly chooses one element from filled part of array to return to the free_list. Fill the gap in O(1) time by moving the bottom element into the gap and shortening the length of the filled part of the array.

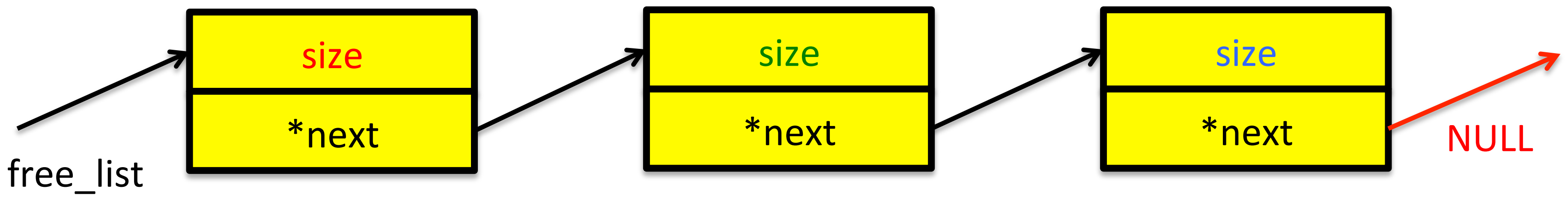
allocatedMemArray



Re-integrate block into free_list

Holds pointers returned by getmem function. Maximum size of n_trials. Fill from position 0 onwards (entire array may not ever be full). The freemem function randomly chooses one element from filled part of array to return to the free_list. Fill the gap in O(1) time by moving the bottom element into the gap and shortening the length of the filled part of the array.

free_list stats

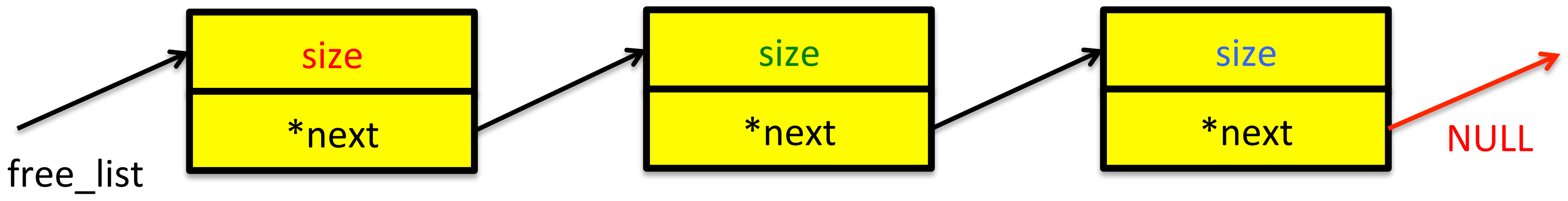


total_free: the total amount of storage in bytes that is currently stored on the free list, including any space occupied by header information or links in the free blocks.

Each block has 16 bytes of header information (size and *next take up 8 bytes each). The value of size in each node is the number of bytes available for storage. So for the above example, total_free is (size + 16) + (size + 16) + (size + 16). Or, traverse the list, adding up all the size fields and counting the list nodes. Add (16 * number of nodes).

n_free_blocks: the total number of individual blocks currently stored on the free list (how many nodes in the list).

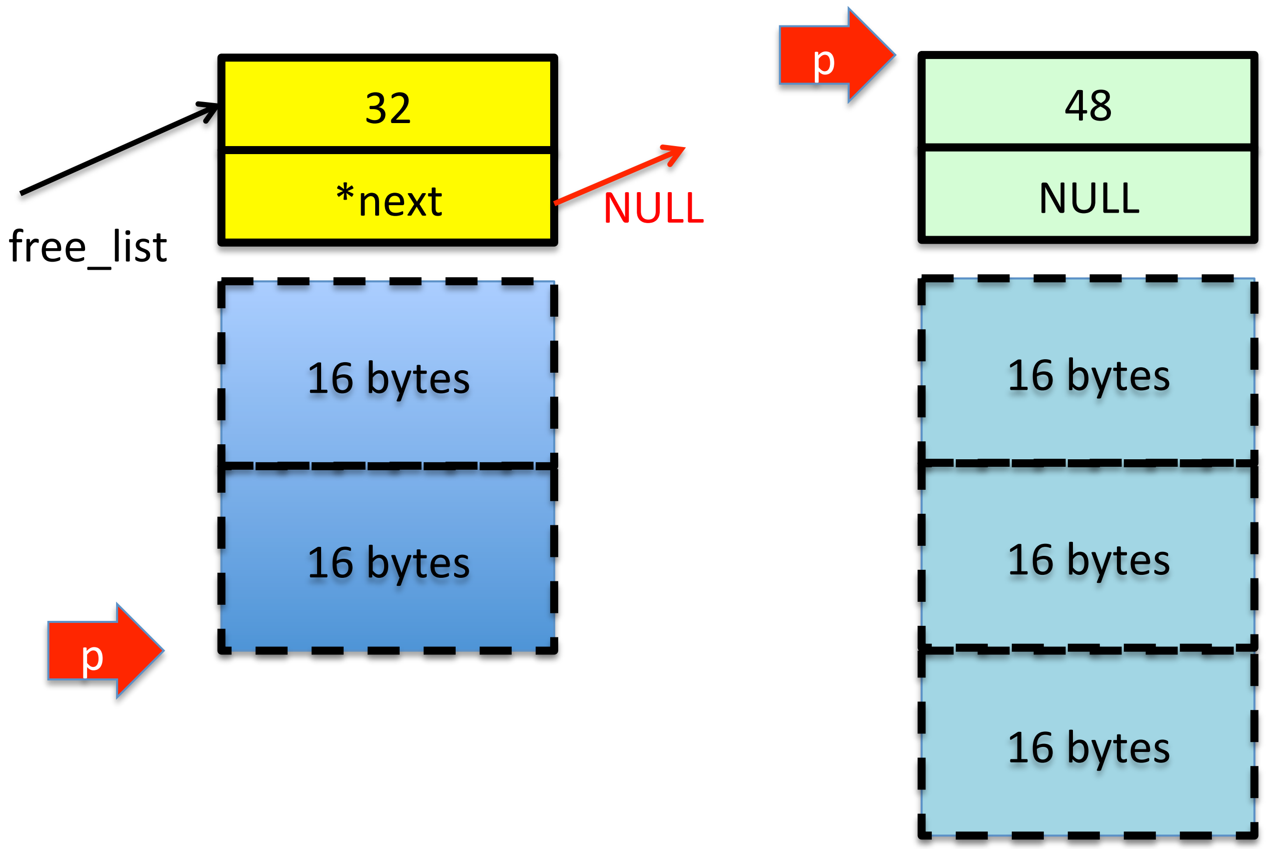
print free_list



print_heap: Each line of output should describe one free block and begin with two hexadecimal numbers (0xdddddddd, where d is a hexadecimal digit) giving the address and length of that block.

Use format %p to print a pointer value, e.g., printf("%p\n", ptr);. For uintptr_t values, since these are stored as long, unsigned integers on our 64-bit systems, they can be printed as decimal numbers using the %lu format specifier: printf("%lu\n",uintvalue);.

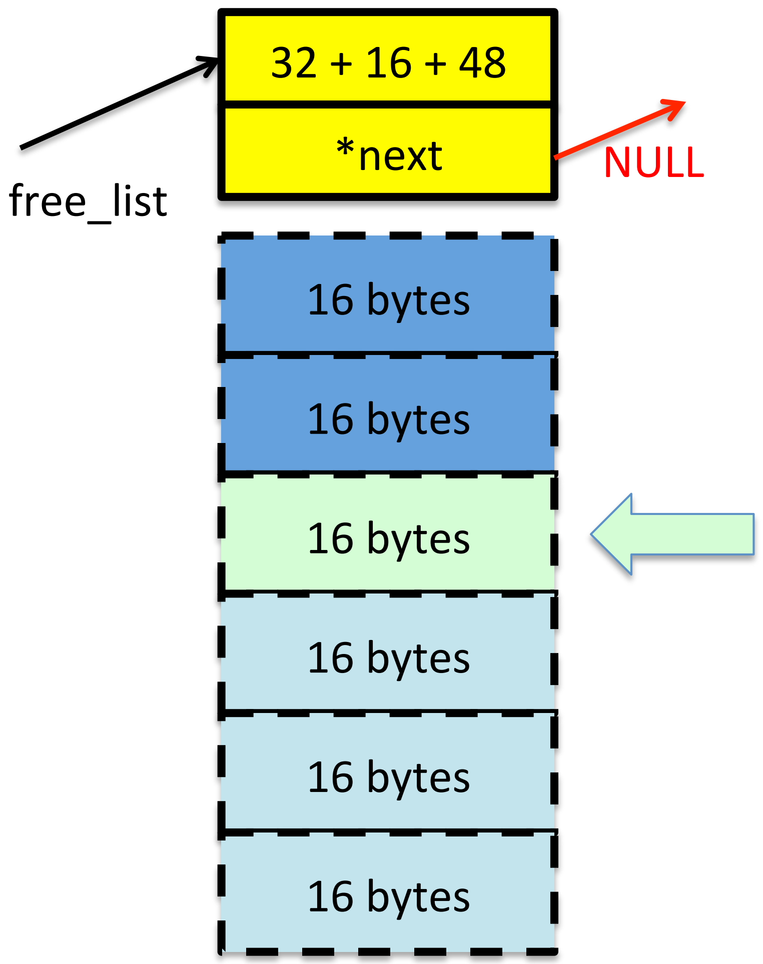
Combining blocks on free_list



When freemem returns a block of storage to the pool, if the block is physically located in memory adjacent to one or more other free blocks, then the free blocks involved should be combined into a single larger block, rather than adding the small blocks to the free list individually.

(just taken off allocatedMemArray)

Combining blocks on free_list



We only need one header per node on the free_list, so use these bytes for storage instead